A Unified Formal Description of Arithmetic and Set Theoretical Data Types

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. We provide a "shared axiomatization" of natural numbers and hereditarily finite sets built around a polymorphic abstraction of bijective base-2 arithmetics.
The "axiomatization" is described as a progressive refinement of Haskell type classes with examples of instances converging to an efficient implementation in terms of arbitrary length integers and bit operations. As an instance, we derive algorithms to perform arithmetic operations efficiently directly with hereditarily finite sets.
The self-contained source code of the paper is available at `http://logic.cse.unt.edu/tarau/research/2010/unified.hs` .
Keywords: *formal description of arithmetic and set theoretical data types, Peano arithmetic and hereditarily finite sets, bijective base-2 arithmetic, software refinement with Haskell type classes, computational mathematics*

1 Introduction

Natural numbers and finite sets have been used as sometimes competing foundations for mathematics, logic and consequently computer science. The de facto standard axiomatization for natural numbers is provided Peano arithmetic. Finite set theory is axiomatized with the usual Zermelo-Fraenkel system (abbreviated ZF) in which the Axiom of Infinity is replaced by its negation. When the axiom of ϵ-induction, (saying that if properties proven on elements also hold on sets containing them, then they hold for all finite sets) is added, the resulting finite set theory (abbreviated ZF^*) is *bi-interpretable* with Peano arithmetic i.e. they emulate each other accurately through a bijective mapping that commutes with standard operations on the two sides ([1]).

This foundational convergence suggests a "shared axiomatization" of Peano arithmetic, hereditarily finite sets and more conventional natural number representations to be used as a unified framework for formally deriving various computational entities.

While axiomatizations of various formal systems are traditionally expressed in classic or intuitionistic predicate logic, equivalent formalisms, in particular the λ-calculus and the type theory used in modern functional languages like Haskell,

can provide specifications in a sometime more readable, more concise, and more importantly, in a genuinely *executable* form.

Our incremental specification loop consists of successive refinements through a chain of Haskell *type classes* (seen as axiom systems) connected by inheritance.

Instances of the type classes (seen as *interpretations* of axiom systems) provide examples that implement various data types in this framework.

The resulting hierarchy of type classes describes incrementally *common computational capabilities* shared by bit-stacks, Peano natural numbers and hereditarily finite sets (sections 3-5).

2 Computing in bijective base-2

Bitstrings provide a common and efficient computational representation for both sets and natural numbers. This recommends their operations as the right abstraction for deriving, in the form of a Haskell type class, a "shared axiomatization" for Peano arithmetic and Finite Set Theory.

While the existence of such a common axiomatization can be seen as a consequence of the bi-interpretability results proven in [1], our distinct executable specification as a Haskell type class provides unique insights into the shared inductive constructions and ensures that computational complexity of operations is kept under control for a variety of instances.

We start by expressing bitstring operations as a Haskell data type:

```
data BitStack = Empty|Bit0 BitStack|Bit1 BitStack
  deriving (Eq, Show, Read)
```

We define the following operations on BitStacks

```
empty = Empty

pushBit0 xs = Bit0 xs
pushBit1 xs = Bit1 xs

popBit (Bit0 xs)=xs
popBit (Bit1 xs)=xs
```

and the predicates

```
empty_ x=Empty==x
bit0_ (Bit0 _)=True
bit0_ _ =False

bit1_ (Bit1 _)=True
bit1_ _=False
```

We remind a few basic (but possibly not widely known) concepts related to the computation mechanism we will use on bitstrings[1].

[1] We assume that bitstrings are mapped to numbers starting with the lowest exponent of 2 and ending with the highest.

Definition 1 *Bijective base-2 representation associates to* $n \in \mathcal{N}$ *a unique string in the regular language* $\{0,1\}^*$ *by removing the 1 indicating the highest exponent of 2 from the standard (complement of 2) bitstring representation of* $n+1$.

Using a list notation for bitstrings this gives: $0 = [], 1 = [0], 2 = [1], 3 = [0,0], 4 = [1,0], 5 = [0,1], 6 = [1,1]$ etc[2].

As a simple exercise in bijective base-2, arithmetic one can now implement the successor function - and therefore provide a model of Peano's axioms, as follows:

```
zero = empty
one = Bit0 empty

peanoSucc xs | empty_ xs = one
peanoSucc xs | bit0_ xs = pushBit1 (popBit xs)
peanoSucc xs | bit1_ xs = pushBit0 (peanoSucc (popBit xs))
```

For instance, 3 applications of `peanoSucc` generate $3 = [0,0]$ as follows:

```
*Unified> (peanoSucc . peanoSucc . peanoSucc) zero
Bit0 (Bit0 Empty)
```

One can verify by structural induction that:

Proposition 1 *Peano's axioms hold with the definition of the successor function provided by* `peanoSucc`.

Using the BitStack representation (by contrast with naive "base-1" successor based definitions), one can implement arithmetic operations like sum and product with low polynomial complexity in terms of the bitsize of their operands. We will defer defining these operations until the next sections, where we will provide such implementations in a more general setting.

Note that as a mild lookahead step towards abstracting away operations on our bitstacks, we have replaced reference to data constructors by the corresponding predicates and functions i.e. `bit0_` `bit1_` etc.

3 Sharing axiomatizations with *type classes*

Haskell's *type classes* [2] are a good approximation of axiom systems as they allow one to describe properties and operations generically i.e. in terms of their action on objects of a parametric type. Haskell's *instances* approximate *interpretations* [1] of such axiomatizations by providing implementations of primitive operations and by refining and possibly overriding derived operations with more efficient equivalents.

We will start by defining a type class that abstracts away the operations on the `BitStack` datatype and provides an axiomatization of natural numbers first, and hereditarily finite sets later.

[2] See `http://en.wikipedia.org/wiki/Bijective_numeration` for the historical origins of the concept and the more general *bijective base-k* case.

3.1 The 5 primitive operations

The class `Polymath` assumes only a theory of structural equality (as implemented by the class `Eq` in Haskell) and the `Read/Show` superclasses needed for input/output.

An instance of this class is required to implement the following 5 primitive operations:

```
class (Eq n,Read n,Show n)⇒Polymath n where
  e :: n
  o_ :: n→Bool
  o :: n→n
  i :: n→n
  r :: n→n
```

We have chosen single letter names `e,o_,o,i,r` for the abstract operations corresponding respectively to `empty, bit0_, pushBit0, pushBit1, popBit` to facilitate a concise "algebraic" view needed to grasp some complex definitions that use compositions of these operations[3].

The `Polymath` type class also provides to its instances generic implementations of the following derived operations:

```
  e_ :: n→Bool
  e_ x = x==e

  i_ :: n→Bool
  i_ x = not (o_ x || e_ x)
```

Note that we use the convention that for each constructor the recognizer's name is obtained by appending "`_`"[4].

While not strictly needed at this point, it is convenient also to include in the `Polymath` type class some additional derived operations. As we will see later, some instances will chose to override them. We first define an object and a recognizer for `1`, the constant function `u` and the predicate `u_`.

```
  u :: n
  u = o e

  u_ :: n→Bool
  u_ x = o_ x && e_ (r x)
```

Next we implement the successor `s` and predecessor `p` functions:

```
  s :: n→n
  s x | e_ x = u
  s x | o_ x = i (r x)
  s x | i_ x = o (s (r x))
```

[3] As an ongoing analogy, the reader can interpret `o` as pushing a `0` to a bitstack, `i` as pushing a `1` and `r` as a pop operation, with `e` representing an empty bitstack.

[4] As part of the bitstack analogy, the predicates `o_` and `i_` can be seen as recognizing respectively a `0` and a `1` (in bijective base-2) at the top of the bitstack.

```
p :: n→n
p x | u_ x = e
p x | o_ x = i (p (r x))
p x | i_ x = o (r x)
```

It is convenient at this point, as we target a diversity of interpretations materialized as Haskell instances, to provide a polymorphic converter between two different instances of the type class `Polymath`. The function `view` allows converting between two different Polymath instances, generically.

```
view :: (Polymath a,Polymath b)⇒a→b
view x | e_ x = e
view x | o_ x = o (view (r x))
view x | i_ x = i (view (r x))
```

3.2 Peano arithmetic

It is important to observe at this point that Peano arithmetic is an instance of the class `Polymath` i.e. that the class can be used to derive an "axiomatization" for Peano arithmetic through a straightforward mapping of Haskell's function definitions to Peano's axioms.

```
data Peano = Zero|Succ Peano deriving (Eq,Show,Read)

instance Polymath Peano where
  e = Zero

  o_ Zero = False
  o_ (Succ x) = not (o_ x)

  o x = Succ (o' x) where
    o' Zero = Zero
    o' (Succ x) = Succ (Succ (o' x))

  i x = Succ (o x)

  r (Succ Zero) = Zero
  r (Succ (Succ Zero)) = Zero
  r (Succ (Succ x)) = Succ (r x)
```

Finally, we can add `BitStack` - which, after all, has inspired the operations of our type class, as an instance of `Polymath`

```
instance Polymath BitStack where
  e=empty
  o=pushBit0
  o_=bit0_
  i=pushBit1
  r=popBit
```

and observe that the Peano and Bitstack interpretations behave consistently:

```
*Unified> i (o (o Empty))
Bit1 (Bit0 (Bit0 Empty))
*Unified> i (o (o Zero))
Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))))
*Unified> i (o (o Empty))
Bit1 (Bit0 (Bit0 Empty))
*Unified> s it
Bit0 (Bit1 (Bit0 Empty))
*Unified> view it :: Peano
Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))))
*Unified> p it
Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))))
Bit1 (Bit0 (Bit0 Empty))
```

Note also the convenience of using `:: view` to instantly morph between instances and the use of Haskell's `it` standing for the previously returned result. So far we have seen that our instances implement syntactic variations of natural numbers equivalent to Peano's axioms. We will now provide an instance showing that our "axiomatization" covers the theory of hereditarily finite sets (assuming, of course, that extensionality, comprehension, regularity, ϵ-induction etc. are implicitly provided by type classes like `Eq` and implementation of recursion in the underlying programming language).

4 Computing with *hereditarily finite sets*

Hereditarily finite sets are built inductively from the empty set (denoted `S []`) by adding finite unions of existing sets at each stage. We first define a rooted tree datatype `S`:

```
data S=S [S] deriving (Eq,Read,Show)
```

To accurately represent sets, the type `S` would require a type system enforcing constraints on type parameters, saying that all elements covered by the definition are distinct and no repetitions occur in any list of type `[S]`. We will assume this and similar properties of our datatypes, when needed, from now on, and consider trees built with the constructor `S` as representing hereditarily finite sets.

We will now show that hereditarily finite sets can do arithmetic as instances of the class `Polymath` by implementing a successor (and predecessor) function. We start with the easier operations:

```
instance Polymath S where
  e = S []

  o_ (S (S []:_)) = True
  o_ _ = False

  o (S xs) = s (S (map s xs))
```

```
i = s . o
```

Note that the `o` operation, that can be seen as pushing a `0` bit to a bitstack is implemented by applying `s` to each branch of the tree. We will now implement `r, s` and `p`.

```
r (S xs) = S (map p (f ys)) where
  S ys = p (S xs)
  f (x:xs) | e_ x = xs
  f xs = xs

s (S xs) = S (hLift (S []) xs) where
  hLift k [] = [k]
  hLift k (x:xs) | k==x = hLift (s x) xs
  hLift k xs = k:xs

p (S xs) = S (hUnLift xs) where
  hUnLift ((S []):xs) = xs
  hUnLift (k:xs) = hUnLift (k':k':xs) where k'= p k
```

First note that *successor* and *predecessor* operations `s,p` are overridden and that the `r` operation is expressed in terms of `p`, as `o` and `i` were expressed in terms of `s`. Next, note that the `map` combinators and the auxiliary functions `hLift` and `hUnlift` are used to delegate work between successive levels of the tree defining a hereditarily finite set.

To summarize, let us observe that the successor and predecessor operations `s,p` at a given level are implemented through iteration of the same at a lower level and that the "left shift" operation implemented by `o,i` results in initiating `s` operations at a lower level. Thus the total number of operations is within a constant factor of the size of the trees.

Let us verify that these operations mimic indeed their more common counterparts on type `Peano`.

```
*Unified> o (i (S []))
S [S [],S [S [S []]]]
*Unified> s it
S [S [S []],S [S [S []]]]
*Unified> view it :: Peano
Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
*Unified> p it
Succ (Succ (Succ (Succ (Succ Zero))))
*Unified> view it :: S
S [S [],S [S [S []]]]
```

It can be proven by structural induction that:

Proposition 2 *Hereditarily finite sets as represented by the data type* `S` *implement the same successor and predecessor operation as the instance* `Peano`.

Note that this implementation of the class `Polymath` implicitly uses the *Ackermann interpretation* of Peano arithmetic in terms of the theory of hereditarily

finite sets, i.e. the natural number associated to a hereditarily finite set is given by the function

$$f(x) = \texttt{if } x = \emptyset \texttt{ then } 0 \texttt{ else } \sum_{a \in x} 2^{f(a)}$$

Let us summarize what's unusual with instance `S` of the class `Polymath`: it shows that successor and predecessor operations can be performed with *hereditarily finite sets playing the role of natural numbers*. As natural numbers and finite ordinals are in a one-to-one mapping, this instance shows that hereditarily finite sets can be seen as *finite ordinals* directly, without using the simple but computationally explosive von Neumann construction (which defines ordinal n as the set $\{0, 1, \ldots, n-1\}$). We will elaborate more on this after defining a total order on our Polymath type.

5 Arithmetic operations

Our next refinement adds key arithmetic operations in the form of a type class extending `Polymath`. We start with addition (`polyAdd`) and subtraction (`polySubtract`):

```
class (Polymath n) ⇒ PolyOrd n where
  polyAdd :: n→n→n
  polyAdd x y | e_ x = y
  polyAdd x y | e_ y = x
  polyAdd x y | o_ x && o_ y =    i (polyAdd (r x) (r y))
  polyAdd x y | o_ x && i_ y = o (s (polyAdd (r x) (r y)))
  polyAdd x y | i_ x && o_ y = o (s (polyAdd (r x) (r y)))
  polyAdd x y | i_ x && i_ y = i (s (polyAdd (r x) (r y)))

  polySubtract :: n→n→n
  polySubtract x y | e_ x && e_ y = e
  polySubtract x y | not(e_ x) && e_ y = x
  polySubtract x y | not (e_ x) && x==y = e
  polySubtract z x | i_ z && o_ x = o (polySubtract (r z) (r x))
  polySubtract z x | o_ z && o_ x = i (polySubtract (r z) (s (r x)))
  polySubtract z x | o_ z && i_ x = o (polySubtract (r z) (s (r x)))
  polySubtract z x | i_ z && i_ x = i (polySubtract (r z) (s (r x)))
```

Efficient comparison uses the fact that with our representation only sequences of distinct lengths can be different. We start by comparing lengths:

```
  lcmp :: n→n→Ordering

  lcmp x y | e_ x && e_ y = EQ
  lcmp x y | e_ x && not(e_ y) = LT
  lcmp x y | not(e_ x) && e_ y = GT
  lcmp x y = lcmp (r x) (r y)
```

Comparison can now proceed by case analysis, the interesting case being when lengths are equal (function `samelen_cmp`):

```
cmp :: n→n→Ordering
cmp x y = ecmp (lcmp x y) x y where
   ecmp EQ x y = samelen_cmp x y
   ecmp b _ _ = b

samelen_cmp :: n→n→Ordering

samelen_cmp x y | e_ x && e_ y = EQ
samelen_cmp x y | e_ x && not(e_ y) = LT
samelen_cmp x y | not(e_ x) && e_ y = GT
samelen_cmp x y | o_ x && o_ y = samelen_cmp (r x) (r y)
samelen_cmp x y | i_ x && i_ y = samelen_cmp (r x) (r y)
samelen_cmp x y | o_ x && i_ y =
  downeq (samelen_cmp (r x) (r y)) where
    downeq EQ = LT
    downeq b = b
samelen_cmp x y | i_ x && o_ y =
  upeq (samelen_cmp (r x) (r y)) where
    upeq EQ = GT
    upeq b = b
```

Finally, boolean comparison operators are defined as follows:

```
lt,gt,eq :: n→n→Bool

lt x y = LT==cmp x y

gt x y = GT==cmp x y

eq x y = EQ==cmp x y
```

After adding the instances

```
instance PolyOrd Peano
instance PolyOrd BitStack
instance PolyOrd S
```

one can see that all operations extend naturally:

```
*Unified> polyAdd (Succ Zero) (Succ Zero)
Succ (Succ Zero)
*Unified> (s.s.s.s) Empty
Bit1 (Bit0 Empty)
*Unified> take 1000 (iterate s (S []))
[S [],S [S []],....,S [S [],S [S [],S [S []]]]]
*Unified> and (zipWith lt it (map s it))
True
```

The last example confirms, for 1000 instances, that we have a *well-ordering* of hereditarily finite sets without recurse to the von Neumann ordinal construction (used in [1] to complete the bi-interpretation from hereditarily finite sets to natural numbers). This replicates a recent result described in [3] where a lexicographic ordering is used to simplify the proof of bi-interpretability of [1].

We will proceed now with introducing more powerful operations. Needless to say, they will apply automatically to all instances of the type class `Polymath`.

6 Adding other arithmetic operations

We first define multiplication.

```
class (PolyOrd n) ⇒ PolyCalc n where
  polyMultiply :: n→n→n
  polyMultiply x _ | e_ x = e
  polyMultiply _ y | e_ y = e
  polyMultiply x y = s (multiplyHelper (p x) (p y)) where
    multiplyHelper x y | e_ x = y
    multiplyHelper x y | o_ x = o (multiplyHelper (r x) y)
    multiplyHelper x y | i_ x = s (polyAdd y  (o (multiplyHelper (r x) y)))

  double :: n→n
  double = p . o

  half :: n→n
  half = r . s
```

Exponentiation by squaring follows - easier for powers of two (`exp2`), then the general case (`pow`):

```
  exp2 :: n→n -- power of 2
  exp2 x | e_ x = u
  exp2 x = double (exp2 (p x))

  pow :: n→n→n -- power y of x
  pow _ y | e_ y = u
  pow x y | o_ y = polyMultiply x (pow (polyMultiply x x) (r y))
  pow x y | i_ y = polyMultiply
    (polyMultiply x x)
    (pow (polyMultiply x x) (r y))
```

After defining instances

```
instance PolyCalc Peano
instance PolyCalc BitStack
instance PolyCalc S
```

operations can be tested under various representations

```
*Unified> polyMultiply (s (s (S []))) (s (s (s (S []))))
S [S [S []],S [S [S []]]]
*Unified> view it :: Peano
Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
*Unified> pow (s (s (S []))) (s (s (s (S []))))
S [S [S [],S [S []]]]
*Unified> view it :: Peano
Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))))
```

7 Deriving set operations

We will now provide a set view of our polymorphic data type. Following [4], where Ackermann's mapping between hereditarily finite sets and natural numbers has been derived as a fold/unfold operation using a bijection between natural numbers and finite sets of natural numbers, we can write:

```
class (PolyCalc n) ⇒ PolySet n where
  as_set_nat :: n→[n]
  as_set_nat n = nat2exps n e where
    nat2exps n _ | e_ n = []
    nat2exps n x = if (i_ n) then xs else (x:xs) where
      xs=nat2exps (half n) (s x)

  as_nat_set :: [n]→n
  as_nat_set ns = foldr polyAdd e (map exp2 ns)
```

Given that natural numbers and hereditarily finite sets, when seen as instances of our generic axiomatization, are connected through Ackermann's bijections, one can shift from one side to the other at will:

```
*Unified> as_set_nat (s (s (s Zero)))
[Zero,Succ Zero]
*Unified> as_nat_set it
Succ (Succ (Succ Zero))
*Unified> as_set_nat (s (s (s (S []))))
[S [],S [S []]]
*Unified> as_nat_set it
S [S [],S [S []]]
```

Note also that, as the operations on type `S` show, the set associated to the number 3 is exactly the same as the first level of its expansion as a hereditarily finite set.

After defining combinators for operations of arity 1 and 2:

```
  setOp1 :: ([n]→[n])→(n→n)
  setOp1 f = as_nat_set . f . as_set_nat
  setOp2 :: ([n]→[n]→[n])→(n→n→n)
  setOp2 op x y = as_nat_set (op (as_set_nat x) (as_set_nat y))
```

we can "borrow" (with confidence!) the usual set operations (provided in the Haskell package Data.List):

```
  setIntersection :: n→n→n
  setIntersection = setOp2 intersect

  setUnion :: n→n→n
  setUnion = setOp2 union

  setDifference :: n→n→n
  setDifference = setOp2 (\\)
```

```
setIncl :: n→n→Bool
setIncl x y = x==setIntersection x y
```

In a similar way, we define a powerset operation conveniently using actual lists, before reflecting it into an operation on natural numbers.

```
powset :: n→n
powset x = as_nat_set
  (map as_nat_set (subsets (as_set_nat x))) where
    subsets [] = [[]]
    subsets (x:xs) = [zs|ys←subsets xs,zs←[ys,(x:ys)]]
```

Next, the ϵ-relation defining set membership is given as the function `inSet`, together with the `augmentSet` function used in various set theoretic constructs as a new set generator.

```
inSet :: n→n→Bool
inSet x y = setIncl (as_nat_set [x]) y

augmentSet :: n→n
augmentSet x = setUnion x (as_nat_set [x])
```

The n-th *von Neumann ordinal* is the set $\{0, 1, \ldots, n-1\}$ and it is used to emulate natural numbers in finite set theory. It is implemented by the function `nthOrdinal`:

```
nthOrdinal :: n→n
nthOrdinal x | e_ x = e
nthOrdinal n = augmentSet (nthOrdinal (p n))
```

Note that as hereditarily finite sets and natural numbers are instances of the class `PolyOrd`, an order preserving bijection can be defined between the two, which makes it unnecessary to resort to von Neumann ordinals to show bi-interpretability [1,3].

After defining the appropriate instances

```
instance PolySet Peano
instance PolySet BitStack
instance PolySet S
```

we observe that set operations act naturally under the hereditarily finite set interpretation:

```
*Unified> (s.s.s.s.s.s)  (S [])
S [S [S []],S [S [S []]]]
*Unified> inSet (S [S []]) it
True

*Unified> powset (S [])
S [S []]
*Unified> powset it
S [S [],S [S []]]
```

```
*Unified> augmentSet (S [])
S [S []]
*Unified> augmentSet it
S [S [],S [S []]]
```

8 Deriving an instance with fast bitstring operations

We will now benefit from our shared axiomatization by designing an instance that takes advantage of bit operations, to implement, through a few overrides, fast versions of our arithmetic and set functions. For syntactic convenience, we will map this instance directly to Haskell's arbitrary length Integer type, to benefit in GHC from the performance of the underlying C-based GMP package. First some arithmetic operations (making use of Haskell's `Data.Bits` library):

```
instance Polymath Integer where
  e = 0
  o_ x = testBit x 0

  o x = succ (shiftL x 1)
  i  = succ . o
  r x | x>0 = shiftR (pred x) 1

  s = succ
  p n | n>0 = pred n
  u = 1
  u_ = (== 1)

instance PolyOrd Integer where
  polySubtract x y = abs (x-y)
  lt = (<)
  polyCompare=compare

instance PolyCalc Integer where
  polyMultiply = (*)
  half x = shiftR x 1
  double x = shiftL x 1
```

Next, some set operations:

```
instance PolySet Integer where
  setUnion = (.|.)
  setIntersection = (.&.)
  setDifference x y = x .&. (complement y)

  inSet x xs = testBit xs (fromIntegral x)

  powset 0 = 1
  powset x = xorL (powset (pred x)) where
    xorL n = n `xor` (shiftL n 1)
```

It is tempting to test for correctness, by computing with the "implementation" provided by the type `Integer` and then reverting to the set view:

```
*Unified> as_nat_set [1,3,4]
26
*Unified> powset it
84215045
*Unified> map as_set_nat (as_set_nat it)
[[],[1],[3],[1,3],[4],[1,4],[3,4],[1,3,4]]
```

It all adds up, but as we do not have a proof yet, we leave it as an *open problem* to show that *the* `xor` *based instance of* `powset` *in* `Integer` *does indeed implement the powerset operation as specified in section 7.*

Finally, we can observe that the von Neumann ordinal construction (used to introduce natural numbers in set theory) defines a fast growing injective function from $\mathcal{N} \to \mathcal{N}$:

```
*Unified> map nthOrdinal [0..4]
[0,1,3,11,2059]
*Unified> as_set_nat 2059
[0,1,3,11]
```

In contrast, our "shared axiomatization" defines ordinals through a trivial *bijection*: the identity function.

Note, as a more practical outcome, that one can now use arbitrary length integers as an efficient representation of hereditarily finite sets. Conversely, a computation like

```
*Unified> s (S [S [S [S [S [S [S [S [S []]]]]]]]])
S [S [],S [S [S [S [S [S [S [S [S []]]]]]]]]]
```

computing easily the successor of a tower of exponents of 2, in terms of hereditarily finite sets, would overflow any computer's memory when using a conventional integer representation.

9 Related work

The techniques described in this paper originate in the data transformation framework described in [5,4,6]. The main new contribution is that while our previous work can be seen as "an existence proof" that, for instance, arithmetic computations can be performed with symbolic objects like hereditarily finite sets, here we show it constructively. Moreover, we lift our conceptual framework to a polymorphic axiomatization which turns out to have as *interpretations* (instances in Haskell parlance) natural numbers, bitstacks and hereditarily finite sets.

Natural number encodings of hereditarily finite sets have triggered the interest of researchers in fields like Axiomatic Set Theory and Foundations of Logic [1,7]. A number of papers of J. Vuillemin develop similar techniques aiming to unify various data types, with focus on theories of boolean functions and arithmetic [8]. Binary number-based axiomatizations of natural number arithmetic

are likely to be folklore, but having access to the the underlying theory of the calculus of constructions [9] and the inductive proofs of their equivalence with Peano arithmetic in the libraries of the `Coq` [10] proof assistant has been particularly enlightening to the author. On the other hand we have not found in the literature any axiomatizations in terms of hereditarily finite sets, as derived in this paper. Future work is planned in proving with Coq the equivalence of operations in Peano arithmetic with their counterparts in the set theoretic interpretation of our type classes.

10 Conclusion

In the form of a literate Haskell program, we have built "shared axiomatizations" of finite arithmetic and hereditarily finite sets using successive refinements of type classes.

We have derived some unusual algorithms, for instance, by expressing arithmetic computations symbolically, in terms of hereditarily finite sets. We have also provided a well-ordering for hereditarily finite sets that maps them to ordinals directly, without using the von Neumann construction.

This has been made possible by extending the techniques introduced in [5,4,6] that allow observing the internal working of intricate mathematical concepts through isomorphisms transporting operations between fundamental data types.

References

1. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
2. Jones, S.P., Jones, M., Meijer, E.: Type classes: An exploration of the design space. In: Haskell Workshop. (1997)
3. Pettigrew, R.: On Interpretations of Bounded Arithmetic and Bounded Set Theory Notre Dame J. Formal Logic Volume 50, Number 2 (2009), 141-151.
4. Tarau, P.: A Groupoid of Isomorphic Data Transformations. In Carette, J., Dixon, L., Coen, C.S., Watt, S.M., eds.: Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009 , Grand Bend, Canada, Springer, LNAI 5625 (July 2009) 170–185
5. Tarau, P.: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In: Proceedings of ACM SAC'09, Honolulu, Hawaii, ACM (March 2009) 1898–1903
6. Tarau, P.: An Embedded Declarative Data Transformation Language. In: Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009, Coimbra, Portugal, ACM (September 2009) 171–182
7. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. **53**(1) (2007) 52–65
8. Vuillemin, J.: Digital algebra and circuits. In Dershowitz, N., ed.: Verification: Theory and Practice. Volume 2772 of Lecture Notes in Computer Science., Springer (2003) 733–746
9. Coquand, T., Huet, G.: The calculus of constructions. Information and Computation **76**(2/3) (1988) 95–120
10. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004) Version 8.0.